# Iterative Methods for Eigenvalue Problems, SVD Calculation, and Applications in Image Compression and Steganography

Torin Kovach
Carnegie Mellon University
Pittsburgh, Pennsylvania
torin@cmu.edu

Andrew Park
Carnegie Mellon University
Pittsburgh, Pennsylvania
ajpark@andrew.cmu.edu

## 1 INTRODUCTION

For the final project of 21-241, we chose to work on building iterative methods to compute eigenvectors and singular vectors. We began by writing programs for the Power Method and QR method to find eigenvectors and eigenvalues, using the Julia language. This included writing our own method for QR decomposition and implementing an algorithm for inverse iteration. Then, we analyzed the performance between the two algorithms for sparse and dense matrices. Next, we implemented each method into a program to calculate singular value decomposition (SVD). Using this SVD calculation program, we used low-rank approximations to compress black and white images. Finally, we used the SVD calculation program to design a steganographic method to encode text into and decode text from black and white images.

### 1.1 Notes On Notation

For the rest of this paper, given matrix $A$, let $A_{*i}$ represent the $i$-th column of $A$, and let $A_{i*}$ represent the $i$-th row of $A$. Furthermore, let $A_{ij}$ or $A_{i,j}$ for more complicated expressions represent the element at the $i$-th row and $j$-th column of $A$.

## 2 POWER METHOD

The power method is an effective method for computing eigenvalues of matrices with special properties. Given $n \times n$ matrix $A$, to use the power method $A$ should have $n$ linearly independent eigenvectors and the eigenvalues, namely $\lambda_i$ for $1 \geq i \geq n$, which can be ordered in magnitude as the following:

$$|\lambda_1| > |\lambda_2| \geq \cdots \geq |\lambda_n|$$

Note that there is the dominant eigenvalue $\lambda_1$, which has the greatest magnitude out of all the eigenvalues of $A$.

We will explain the process and motivation behind the power method. Let $\vec{x}_0$ be an arbitrary vector of length $n$, and let $\{\vec{v}_1, \vec{v}_2, \ldots, \vec{v}_n\}$ be a set of $n$ linearly independent eigenvectors of $A$. As $A$ is $n \times n$ and has $n$ distinct eigenvectors, it follows that there must exist a $\{a_1, a_2, \ldots, a_n\} \subseteq \mathbb{R}$ such that $\vec{x}_0$ can be written as:

$$\vec{x}_0 = a_1\vec{v}_1 + a_2\vec{v}_2 + \cdots + a_n\vec{v}_n$$

From this equation, we can express $A^n\vec{x}_0$ for an arbitrary $m \in \mathbb{N}^+$.

$$\begin{aligned} A\vec{x}_0 &= a_1\lambda_1\vec{v}_1 + a_2\lambda_2\vec{v}_2 + \cdots + a_n\lambda_n\vec{v}_n \\ A^2\vec{x}_0 &= a_1\lambda_1^2\vec{v}_1 + a_2\lambda_2^2\vec{v}_2 + \cdots + a_n\lambda_n^2\vec{v}_n \\ &\cdots \\ A^m\vec{x}_0 &= a_1\lambda_1^m\vec{v}_1 + a_2\lambda_2^m\vec{v}_2 + \cdots + a_n\lambda_n^m\vec{v}_n \end{aligned}$$

By dividing by $\lambda_1^m$, we find:

$$\frac{A^m\vec{x}_0}{\lambda_1^m} = a_1\vec{v}_1 + a_2\left(\frac{\lambda_2}{\lambda_1}\right)^m\vec{v}_2 + \cdots + a_n\left(\frac{\lambda_n}{\lambda_1}\right)^m\vec{v}_n$$

Because $\lambda_1$ is the dominant eigenvalue, we know that $|\lambda_i/\lambda_1| < 1$ for all $1 < i \leq n$. As a result, the the terms $\lambda_2/\lambda_1, \lambda_3/\lambda_1, \cdots \lambda_n/\lambda_1$ each approach zero as $m$ increases. Thus, for sufficiently large $m$:

$$\frac{A^m\vec{x}_0}{\lambda_1^m} \approx a_1\vec{v}_1 + a_2(0)\vec{v}_2 + \cdots + a_n(0)\vec{v}_n = a_1\vec{v}_1$$

Now, if we divide the $(m+1)$-th term by the $m$-th term, we can find an approximation for the dominant eigenvalue:

$$\frac{A^{m+1}\vec{x}_0}{A^m\vec{x}_0} = \frac{\lambda_1^{m+1}a_1\vec{v}_1}{\lambda_1^m a_1\vec{v}_1} = \lambda_1$$

In order to find the rest of the eigenvalues, we can simply project the matrix A onto the orthogonal space of the largest eigenvalue, in a process known as deflation. Then, we can apply the power method again to the new, smaller matrix to find the next biggest eigenvalue. As a result, we can recursively find all $n$ eigenvalues of our given matrix $A$.

## 3 QR METHOD

The QR method is the second eigenvalue algorithm we discuss in this paper. It is premised on repeatedly performing QR decomposition, multiplying the resulting matrices in reverse order, performing QR decomposition again, and onwards to produce an upper triangular matrix with the eigenvalues on its diagonal.

### 3.1 QR Decomposition

While the Julia language includes a functionality to perform QR decomposition, as this process is such a primary part of the QR method, we endeavored to design our own program to calculate QR decomposition from scratch. This was done in two steps – firstly, we wrote a naive QR decomposition algorithm, and secondly, we improved this algorithm to produce a computationally faster one. Both QR decomposition algorithms start with an $m \times n$ matrix $A$ with linearly independent columns (as a result, $m \geq n$), and produce $m \times n$ orthonormal matrix $Q$ and $n \times n$ upper triangular matrix $R$ such that $A = QR$. The naive algorithm proceeds using this simple recursive definition for $1 \leq i \leq m$:

$$\vec{w}_i = A_{*i} - \sum_{k=1}^{i-1} \text{proj}_{Q_{*k}} A_{*i} \qquad Q_{*i} = \frac{\vec{w}_i}{|\vec{w}_i|}$$

$$R_{ij} = Q_{*i}^T A_{*j}$$

As discussed in class, this is simply the Gram-Schmidt algorithm – we treat the columns of $A$ as a basis of a subspace in $\mathbb{R}^m$, and the columns of $Q$ as an orthonormal basis of the same subspace. Note that for $1 \leq j \leq n$, $A_{*j}$ is orthogonal to $Q_{*i}$ for $j < i \leq n$. Thus, for all $i > j$, $R_{ij} = 0$, and as a result $R$ is upper triangular. This method directly transcribed to code produces the algorithm:

**Algorithm 1** Naive $QR$ Decomposition

**Require:** $A$ is $m \times n$, has independent columns
**Ensure:** $Q$ orthonormal, $R$ upper triangular, $A = QR$
1: $Q \leftarrow m \times n$ matrix
2: $R \leftarrow n \times n$ matrix, all values 0
3: **for** $i$: $1 \rightarrow n$ **do**
4: $\quad w \leftarrow A_{*i}$
5: $\quad$ **for** $k$: $1 \rightarrow i-1$ **do**
6: $\quad\quad w \leftarrow w - (Q_{*k}^T A_{*i}) Q_{*k}$
7: $\quad$ **end for**
8: $\quad Q_{*i} = w \,/\, ||w||$
9: **end for**
10: **for** $i$: $1 \rightarrow n$, $j$: $1 \rightarrow m$ **do**
11: $\quad R_{ij} \leftarrow Q_{*i}^T A_{*j}$
12: **end for**
13: **return** $Q, R$

Note that as the columns of $Q$ are orthonormal, the formula for projections can be simplified to that used on line 6.
Now, we move on to our more efficient, modified method for QR decomposition. Note that as $A = QR$, $R$ upper triangular, it follows that for $1 \leq i \leq n$:

$$A_{*i} = \sum_{k=1}^{i} Q_{*k} R_{ki} \implies A = \sum_{k=1}^{n} Q_{*k} R_{k*}$$

From this summation, we can define a series $A^{(i)}$ for $1 \leq i \leq n+1$:

$$A^{(i)} = \sum_{k=i}^{n} Q_{*k} R_{k*}$$

This summation will give us a few interesting properties. First, where $\vec{e}_i$ is a standard basis vector, it is possible to show that:

$$\begin{aligned} A_{*i}^{(i)} &= A^{(i)} \vec{e}_i \\ &= \sum_{k=i}^{n} (Q_{*k} R_{k*} \vec{e}_i) \\ &= Q_{*i} R_{ii} \end{aligned}$$

As a result, $Q_{*i} = \frac{A_{*i}^{(i)}}{R_{ii}}$, and as $Q$ orthonormal, $R_{ii} = ||A_{*i}^{(i)}||$. Furthermore, using the fact that $Q$ is orthonormal we can show:

$$\begin{aligned} \left(A^{(i)}\right)^T Q_{*i} &= \left(\sum_{k=i}^{n} Q_{*k} R_{k*}\right)^T Q_{*i} \\ &= \sum_{k=i}^{n} R_{k*}^T Q_{*k}^T Q_{*i} \\ &= R_{i*}^T \end{aligned}$$

As a result, $R_{ki} = \left(A_{*k}^{(i)}\right)^T Q_{*i}$. Thus, using $A^{(i)}$, we can find $R_{*i}$. Now, note that $A^{(1)} = A$. From this fact, we can show that:

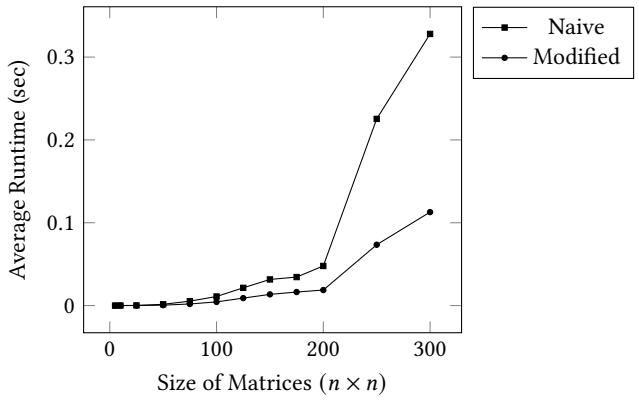
$$A^{(i)} - Q_{*i} R_{i*} = \sum_{k=i+1}^{n} Q_{*k} R_{k*} = A^{(i+1)}$$

Thus, we can iteratively calculate $i+1$. Using this process, we can define a new, more efficient $QR$ algorithm.

**Algorithm 2** Modified $QR$ Decomposition

**Require:** $A$ is $m \times n$, has independent columns
**Ensure:** $Q$ orthonormal, $R$ upper triangular, $A = QR$
1: $Q \leftarrow m \times n$ matrix
2: $R \leftarrow n \times n$ matrix, all values 0
3: **for** $i$: $1 \rightarrow n$ **do**
4: $\quad R_{ii} \leftarrow ||A_{*i}||$
5: $\quad w \leftarrow A_{*k} \,/\, R_{ii}$
6: $\quad Q_{*i} \leftarrow w \,/\, ||w||$
7: $\quad$ **for** $j$: $i+1 \rightarrow n$ **do**
8: $\quad\quad R_{ij} = A_{*j}^T Q_{*i}$
9: $\quad\quad A_{*j} \leftarrow A_{*j} - (R_{ij}) Q_{*i}$
10: $\quad$ **end for**
11: **end for**
12: **return** $Q, R$

To demonstrate the difference in efficiency between our algorithms, we plotted their runtimes to calculate QR decompositions of symmetric matrices. The figure below documents the average number of seconds each algorithm took to find $QR$ decompositions for 100 symmetric matrices of various sizes.




We see quite clearly our efforts were not in vain – the modified algorithm runs much faster!

## 3.2 QR Method

Now that we have code to calculate a $QR$ decomposition, we have the necessary tools to write our eigenvalue algorithm. Given matrix $A$, consider a sequence of matrices $(A_k)_{k \in \mathbb{N}}$, with $A = A_0$. Given $QR$ decomposition for $i \in \mathbb{N}$ as $A_i = Q_i R_i$, our sequence is defined as $A_{i+1} = R_i Q_i$. By the fact that $Q$ orthonormal, it follows:

$$A_{i+1} = Q_i^T Q_i A_{i+1} = Q_i^T Q_i R_i Q_i = Q_i^T A_i Q_i$$

Thus, $A_{i+1}$ and $A_i$ are similar, and as a result they must have the same eigenvalues. Inductively, it follows that all of the matrices in our sequence have the same eigenvalues. The $QR$ algorithm can be viewed as similar to the "power" method – however, instead of working upon single vectors, it works on a basis of vectors. As we continue the $QR$ iteration, we find that $A$ slowly becomes upper triangular – as a result, the eigenvalues of $A$ appear on the diagonal of $A_i$. To actually implement the algorithm, we simply iterated through the matrix until the all values on the diagonal converged within a certain error bound. The algorithmic implementation is shown at the start of page 3.

**Algorithm 3** Naive $QR$ Method

**Require:** $A$ is $n \times n$, $M$ is reasonable margin of error
**Ensure:** $(\forall \lambda \in L)(\exists \vec{v} \in \mathbb{R}^n)(A\vec{v} \approx \lambda\vec{v})$
$L \leftarrow$ array of size $n$
$i \leftarrow n$ *(indexing current eigenvalue)*
$c \leftarrow A_{ii}$ *(current eigenvalue approximation)*
**while** $i > 0$ **do**
  $Q, R \leftarrow QR$ decomp. of $A$
  $A \leftarrow RQ$ *(calculating next item in sequence)*
  **if** $|A_{ii} - c| \leq M$ **then**
    $L \leftarrow A_{ii}$
    $i \leftarrow i - 1$
  **else**
    $c = A_{ii}$
  **end if**
**end while**
**return** $L, V$

While this algorithm provides accurate eigenvalues, its complexity is still impractical. We first implemented a couple key modifications to hasten our algorithm. Then, we implemented a few more key modifications to ensure accurate eigenvalue calculation.

*Deflation.* Through the $QR$ iteration, we calculate eigenvalues iteratively from $n$ to 1. As a result, once we have calculated the $i$-th eigenvalue, it is not necessary to retain the $i$-th through $n$-th columns to produce the rest of the eigenvalues. As a result, once we have converged upon a specific eigenvalue, we can continue computation with a smaller matrix with the eigenvalue removed. While this does not have a sizable impact at the beginning of the $QR$ algorithm, it greatly speeds up the last iterations of the algorithm, as we operate with increasingly small matrices.

*Shifting.* Note that if we can cause the $QR$ iteration to converge faster, it will result in greater efficiency. In our first implementation, the speed of convergence is approximately the ratio between eigenvalues, that is $|\lambda_{i-1} \,/\, \lambda_i|$ when we are attempting to find eigenvalue $\lambda_i$ (smaller ratio results in faster convergence). One method to hasten convergence is to implement a type of shifting – that is, given a sequence $(\mu_k)_{k\in\mathbb{N}}$, we modify our method so that:

$$A_i - \mu_i I = Q_i R_i \qquad A_{i+1} = R_i Q_i + \mu_i I$$

With a few simple computations, we see that the similarity relationship between $A_i$ and $A_{i+1}$ is still preserved with this new formula:

$$\begin{aligned} A_{i+1} - \mu_i I &= R_i Q_i \\ &= Q_i^T Q_i R_i Q_i \\ &= Q_i^T (A_i - \mu_i I) Q_i \\ &= Q_i^T A_i Q_i - \mu_i I \\ \implies \quad A_{i+1} &= Q_i^T A_i Q_i \end{aligned}$$

From this shift, the ratio from which we can determine the speed of converge is effectively $|(\lambda_{i-1} - \mu) \,/\, (\lambda_i - \mu)|$, where $\mu$ is the current value in our sequence. To minimize this ratio, we aim to select a $\mu$ close to $\lambda_{i-1}$. One effective method is to set $\mu = (A_i)_{nn}$, known as shifting by the *Rayleigh quotient*. Another method, used specifically for symmetric matrices, is to calculate a *Wilkinson Shift*. Given that the lower right sub-matrix of $A_i$ is $[x, y; y, z]$, we determine $\mu_i$ as:

$$\delta = \frac{x - z}{2} \qquad \mu_i = z - \frac{\operatorname{sign}(\delta) \cdot y^2}{|\delta| - \sqrt{\delta^2 + y^2}}$$

Note that $\operatorname{sign}(\delta)$ equals $-1$ or 1, and is arbitrarily equal to 1 if $\delta = 0$. We implemented and tested both of these methods.

## 3.3 Eigenvector Calculation

While implementing shifting or deflation allowed us to calculate extremely precise eigenvalues, we still have yet to calculate eigenvectors. Our solution to this problem took the form of two steps – firstly, we added elements of simultaneous iteration, and secondly, we implemented inverse iteration.

Earlier we stated that the $QR$ method can be thought of as the power method operating on a basis of vectors. We will elaborate more on this idea now. Suppose we chose a basis of starting vectors for the power method to operate on $n \times n$ matrix $A$, namely $\vec{v}_1, \vec{v}_2, \ldots, \vec{v}_n$. First, we impose these vectors into the columns of a matrix $V$. Now, for large $k$, we simply take $A^k V$ producing a basis of eigenvectors. However, these eigenvectors may all be the exact same as one another. So, instead of simply taking a power, we iteratively orthogonalize our basis $V$ using QR decomposition. This presents a recursive formula as shown for $k \in \mathbb{N}^+$, known as simultaneous iteration:

$$V_0 = Q_0 R_0, \quad W_k = A Q_{k-1}, \quad W_k = Q_k R_k$$

All $Q_k$ are orthonormal, and all $R_k$ are upper triangular. As a result, as we iterate through new values of $W$, we generally find that $Q_k$ will result in a basis of unit eigenvectors of $A$. We can think of the QR method as a simultaneous iteration process, starting with $V = I$. In terms of the $QR$ method, let $\hat{Q}_i$ be defined as:

$$\hat{Q}_i = \prod_{k=0}^{i} Q_i = Q_0 Q_1 \ldots Q_i$$

That is, $\hat{Q}_i$ is equal to the product of the values for $Q$ for each $QR$ decomposition when conducting the $QR$ method. Using this new definition, we can rewrite the $QR$ method similarly to simultaneous iteration:

$$\hat{Q}_0 = I, \quad W_k = A\hat{Q}_{k-1}, \quad W_k = \hat{Q}_k R_k, \quad A_k = \hat{Q}_k^T A \hat{Q}_k$$

To see more clearly why $A_k = \hat{Q}_k^T A \hat{Q}_k$, note that $A_{k+1} = Q_k^T A_k Q_k$, and as a result $A_1 = Q_0^T A Q_0$. Proceeding inductively, it follows that:

$$A_k = Q_k^T \dots Q_1^T Q_0^T A Q_0 Q_1 \dots Q_k \implies A_k = \hat{Q}_k^T A \hat{Q}_k$$

Note that as the similarity relation $A_{k+1} = Q_k^T A_k Q_k$ is preserved when shifting, thinking about the $QR$ method as simultaneous iteration remains valid when shifting is also implemented.

The simultaneous iteration sequence defined above and our traditional $QR$ method will produce the same sequences of $A_k$, $\hat{Q}_k$. However, now we can use simultaneous iteration to conclude that $\hat{Q}_k$ will generally converge to a basis of eigenvectors for $A$.

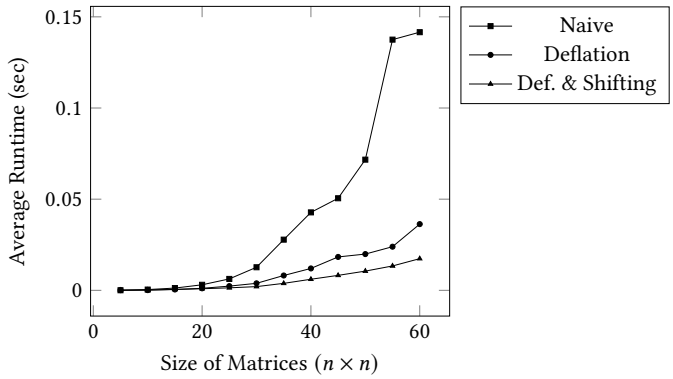
In practice, this method may often require far more iterations to generate eigenvectors as opposed to simply generating eigenvalues. Furthermore, the eigenvalue approximations proved more accurate. As a result, we implemented a secondary method to use on the eigenvectors produced by our $QR$ method to ensure accurate calculations. This method is known as inverse iteration.

Inverse iteration, or the inverse power method, is very similar to the power method. Given an approximation of an eigenvector $\vec{b}_0$, and an approximation of corresponding eigenvalue $\mu$, we can iteratively find a better approximation for the eigenvector as so:

$$\vec{v}_i = (A - \mu I)^{-1} \vec{b}_i \qquad \vec{b}_{i+1} = \frac{\vec{v}_i}{||\vec{v}_i||}$$

While the power method takes powers of $A^k$, the inverse power method instead takes powers of $(A^{-1})^k$. The $\mu$ simply determines our shift. For each eigenvector/eigenvalue pair produced by the QR method, we did a few iterations of the inverse power method. After just a few iterations, our approximations for the eigenvectors of $A$ become extremely accurate. Our final eigenvector/eigenvalue method works as follows:

To compare the efficiencies of our naive QR method, our QR method with deflation, and our QR method with shifting (Rayleigh shift) and deflation, we computed runtimes to generate eigenvalues averaged over 100 $n \times n$ symmetric matrices of varying $n$. This is displayed in the figure below:




Interestingly, for smaller matrices ($5 \leq n \leq 15$), the deflation-only algorithm has a lower runtime than the algorithm implementing both deflation and shifting. This could be attributed to the overhead required for shifting, which presents a net benefit only when we are working with large enough matrices.

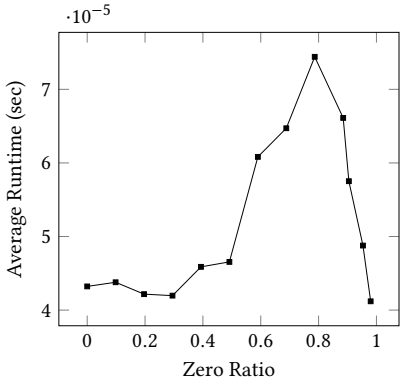
**Algorithm 4** Final $QR$ Method

**Require:** $A$ is $n \times n$, $M$ is reasonable margin of error, $k \in \mathbb{Z}^+$
**Ensure:** elements of $L$ are valid eigenvalues of $A$
**Ensure:** columns of $\hat{Q}$ are valid eigenvectors of $A$

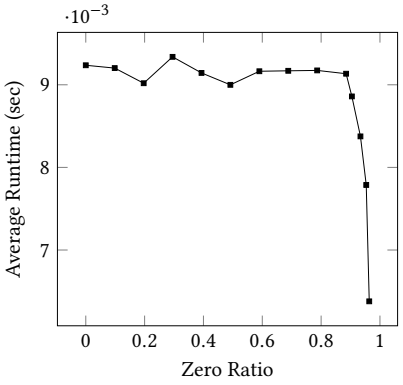
$L \leftarrow$ array of size $n$
$\hat{Q} \leftarrow I_n$
$i \leftarrow n$ *(indexing current eigenvalue)*
$c \leftarrow A_{ii}$ *(current eigenvalue approximation)*
**while** $i > 0$ **do**
  $\mu \leftarrow$ shift value (i.e. $A_{nn}$)
  $Q, R \leftarrow QR$ decomp. of $A - \mu I$
  $\hat{Q} \leftarrow \hat{Q}Q$
  $A \leftarrow RQ + \mu I$ *(calculating next item in sequence)*
  **if** $|A_{ii} - c| \leq M$ **then**
    $L \leftarrow A_{ii}$
    $i \leftarrow i - 1$
  **else**
    $c = A_{ii}$
  **end if**
**end while**
**for** $i : 1 \rightarrow n$, $j : 1 \rightarrow k$ **do**
  $v = (A - L[i] \cdot I)^{-1} \hat{Q}_{*i}$
  $\hat{Q}_{*i} = v \,/\, ||v||$
**end for**
**return** $L, \hat{Q}$

## 4 SPARSE AND DENSE MATRICES

Often the density of a matrix can affect the speed at which our eigenvalue methods can work. To investigate this, we built and conducted an experiment to test the performance of each method on sparse and dense matrices of the same size. To measure how sparse or dense a matrix was, we calculated a "zero ratio," or the ratio between the number of zeros in the matrix and the total entries in the matrix. For a range of varying zero ratios, we randomly generated sample sets of 100 symmetric matrices of approximately the ratio (±2%), and found the average runtime for each algorithm to find eigenvalues for each matrix. As our implementation of the power method was somewhat limited, we instead used the power method provided by the Julia IterativeSolvers package for testing. Displayed below is the averaged runtimes for the power method:



The figure above shows that the power method has a far greater runtime for matrices of approximately 75% zeros, but runtime quickly decreases after this point as matrices become more and more sparse. In contrast, we can see the graph implemented for the QR method (with deflation and a Rayleigh shift) below.




Note that this figure is at a magnitude of 100 times that of the figure with the power method. While this might imply that the power method is far faster than the QR method for this size of matrix, we posit that this difference is simply an artifact of the power method being designed professionally for the Julia language, while the QR algorithm was put together from scratch by ourselves. We see that while the QR method remains relatively constant in runtime for almost all levels of density, only when matrices become very dense (density ratio > 90%) does runtime begin to decrease.

The fact that the QR method retains similar runtime for matrices of high or medium density, while the power method shows a steep increase in runtime for matrices of medium density, suggests that the QR method would be preferable for dense matrices. Furthermore, the fact that the power method begins a decrease in runtime for increasingly sparse matrices earlier than the QR method supports the idea that the power method would be preferable for use with sparse matrices.

## 5 SINGULAR VALUE DECOMPOSITION

The singular value decomposition (SVD) of a matrix $m \times n$ matrix $A$ takes the form of $A = U\Sigma V^T$. Where $U$ is $m \times m$, $\Sigma$ is $m \times n$ and $V$ is $n \times n$. Furthermore, the columns of $U$ and the columns of $V$ are orthonormal and $\Sigma$ has value 0 on every element except those on its diagonal, which can be 0 or nonzero. It follows that the columns of $V$ are the eigenvectors of $A^TA$, the columns of $U$ are the eigenvectors of $AA^T$. Finally, the non-zero values upon the diagonal of $\Sigma$, namely $\sigma_1, \sigma_2, \ldots, \sigma_r$ are the square root of the shared eigenvalues between $AA^T$ and $A^TA$. The rest of the singular values, $\sigma_{r+1}, \ldots, \sigma_{\min(m,n)}$ are zero. Thus, for $1 \leq i \leq \min(m, n)$, it follows:

$$AV_{*i} = \sigma_i U_{*i} \quad \text{and} \quad A^T U_{*i} = \sigma_i V_{*i}$$

Thus, either by calculating the unit eigenvectors/eigenvalues of $A^TA$ or calculating the unit eigenvectors/eigenvalues of $AA^T$, we can determine the singular value decomposition of our matrix. This is summarized in the algorithm below:

**Algorithm 5** SVD Calculation

**Require:** $A$ is $m \times n$
**Ensure:** $A = U\Sigma V^T$
**Ensure:** $U, \Sigma, V$ in proper $SVD$ format
$\Sigma \leftarrow m \times n$ matrix
**if** $m \leq n$ **then**
  $L, V \leftarrow$ get eigenvalues/vectors $(L, \hat{Q})$ of $A^TA$
  $U \leftarrow m \times m$ matrix
  **for** $i$: $1 \rightarrow m$ **do**
    $\Sigma_{ii} \leftarrow \sqrt{(L[i])}$
    $U_{*i} = (AV_{*i}) \,/\, \Sigma_{ii}$
  **end for**
**else**
  $L, U \leftarrow$ get eigenvalues/vectors $(L, \hat{Q})$ of $AA^T$
  $V \leftarrow n \times n$ matrix
  **for** $i$: $1 \rightarrow n$ **do**
    $\Sigma_{ii} \leftarrow \sqrt{(L[i])}$
    $V_{*i} = (A^TU_{*i}) \,/\, \Sigma_{ii}$
  **end for**
**end if**
**return** $U, \Sigma, V$

## 6 IMAGE COMPRESSION

As discussed in class, we can use singular value decomposition to produce a compressed version of an image. For the sake of simplicity, we simply use black and white images. In this way, images can be represented as a matrix with each value representing the "darkness" at each pixel position. This compression is a result of us being able to write out a singular value decomposition not just as the product of three matrices, but as the sum of multiple vector products:

$$A = \sum_{i=1}^{r} \sigma_i U_{*i} V_{*i}^T$$

It follows that we can form a low rank approximation by removing elements of the summation with low values for $\sigma_i$. As the singular values are listed in the SVD from greatest to least, we can quite efficiently use low rank approximation to create an image compression by setting singular values to 0 (effectively shortening the summation).

Note that singular value decomposition image compression, while effective in producing an accurate result, is very memory and computation intensive. Furthermore, as SVD compression has a floating-point implementation, it presents multiple floating-point problems compared to more often-used fixed-point algorithms. However, SVD image compression is an effective demonstration of the SVD. To show a practical test of our algorithm, we computed the compression for some stock images provided by Julia. Specifically, the image presented here is originally $512 \times 512$.

From left to right then top to bottom, the images above are the original image, a rank 200 compression, rank 100 compression, rank 50 compression, rank 30 compression, and finally a rank 5 compression. As you can see, even at relatively row ranks, we can still produce an image with very low loss of image quality.

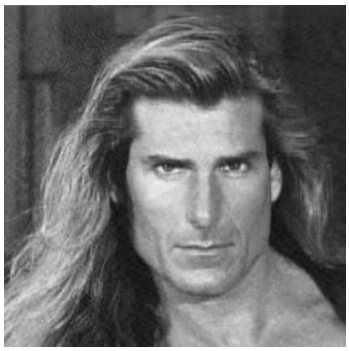
## 7 IMAGE STEGANOGRAPHY

Using singular value decomposition, we can also implement image steganography for a black and white image. Steganography is the process of concealing information within a file, message, video, or in our case, an image. Generally, we aim to hide data in parts of our image file that result in the least change to end output (how the picture looks). Fortunately, SVD provides a simple way to classify what will create the most change to our output and what will create the least. Similarly to how low rank approximations remove data that is the least important to a final image, our steganographic algorithm modifies this data to contain our message.

First, we start with a string of zeroes and ones that we would like to hide in our image. For example, we could take the string “torin" converted from ASCII to binary:

011101000110111101110010011010010110111000000000

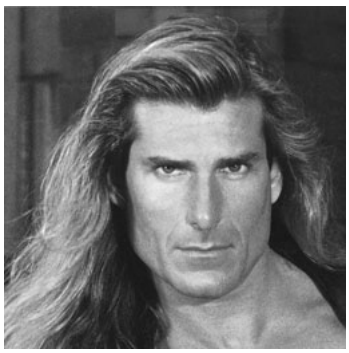
This results in bit-string of 48 total elements (we end with a null-terminator, "\0"). Next, we find an image to hide our message in. For our example, we will use a 256x256 pixel stock image of Fabio provided by Julia:



Julia allows us to convert this image into a matrix $M$ of float values, of dimension $256 \times 256$. It follows that we can partition $M$ into a block matrix containing 1024 smaller matrices, each with dimension $8 \times 8$. We will encode a single 0 or 1 into each of the first 48 of these matrices.

Now, consider arbitrary $8 \times 8$ matrix $A$ from the block matrix $M$. It follows that by taking the SVD, we find $A = U\Sigma V^T$. Let the singular values, or those values along the diagonal of $\Sigma$, be $\sigma_1, \sigma_2, \ldots, \sigma_8$. By nature of SVD, we know that the last few singular values have a very small impact on the final matrix $A$, while the first few singular values have a much greater impact on the final matrix $A$. As a result, we will use the last few singular values to transmit our message. If we would like to transmit a 0, we will modify $A$ so that $\sigma_8 \gg 0$. If we would like to transmit a 1, we will modify $A$ so that $\sigma_8 \approx 0$.

To modify our matrix $A$ so that $\sigma_8 \approx 0$, we can simply set $\sigma_8 = 0$. However, to modify our matrix $A$ so that $\sigma_8 \gg 0$, we can not simply set $\sigma_8 = 0.01$ (or some other arbitrary constant of a reasonable magnitude). This will be ineffective, if, for instance $\sigma_7 = 0$. Thus, for each value $\sigma_1$ through $\sigma_8$, if the value is very close to 0, we set it to a non-zero constant. In this way, we can ensure that when the decoder recalculates the SVD, $\sigma_8 \approx 0$. Once we have set our new singular values, we simply recalculate $A_c = U\Sigma V^T$, where $A_c$ is our encoded matrix. By calculating $A_c$ for the first 48 block matrices and imposing them back into $M$, we have our encoded image $M_c$. For our example, the resulting image is shown below.



As you can see, the message is hidden with imperceptible difference in the final image. To decrypt the encoded image, one simply

continues to take SVD of the smaller matrices and measure $\sigma_8$ until they reach the null-terminator.

## ACKNOWLEDGMENTS

This work was completed as a final project for the 21-241 class for the Fall 2020 semester. We thank the teaching staff, including teaching assistants and professor David Offner, for contributing their time and expertise.

## REFERENCES

Panju, M. (2011). Iterative Methods for Computing Eigenvalues and Eigenvectors. The Waterloo Mathematics Review. http://mathreview.uwaterloo.ca/archive/voli/1/panju.pdf

Saad, Y. (2011). Numerical methods for large eigenvalue problems (Rev. ed.). Society for Industrial and Applied Mathematics SIAM, 3600 Market Street, Floor 6, Philadelphia, PA 19104. https://epubs.siam.org/doi/book/10.1137/1.9781611970739

Strang, G. (2016). Introduction to Linear Algebra (Gilbert Strang) (5th ed.). Wellesley-Cambridge Press. https://math.mit.edu/ gs/linearalgebra

The QR algorithm. (n.d.). Retrieved December 11, 2020, from http://pi.math.cornell. edu/ web6140/TopTenAlgorithms/QRalgorithm.html

Wengerhoff, D. (2006). Using the singular value decomposition for image steganography. ProQuest Dissertations Publishing. https://lib.dr.iastate.edu/cgi/viewcontent.cgi?article=2387context=rtd

White, P. A., & Brown, R. R. (1964). A comparison of methods for computing the eigenvalues and eigenvectors of real symmetric matrix. Mathematics of Computation, 18(87), 457. https://doi.org/10.1090/s0025-5718-1964-0165667-0

# Final Code

December 11, 2020

This Notebook contains the breadth of code that we wrote and implemented in this project. It assumes all necessary packages have already been added.

[1]:
```
using Polynomials
using LinearAlgebra
using Statistics
using Random
using IterativeSolvers
#For image compression, image steganorgraphy:
using Images
using TestImages
using Colors
```

**Power Method**

[2]:
```
# Calculates the largest eigenvalue via the power method
function max_power_method(A)
    y_final = fill(1, size(A)[1])
    tol=1e-11 #Margin of error to measure convergence
    m = 0
    while(true)
        mold = m
        #Save old eigenvector
        y_old = y_final
        y_final = A * y_final
        m = dot(y_final, y_old)/dot(y_old, y_old)
        scale = maximum(abs.(y_final))
        y_final = y_final/scale
        #Check for convergence
        if (abs(m - mold) < tol)
            return m, y_final
        end
    end
end
```

[2]:
```
max_power_method (generic function with 1 method)
```

[3]:
```
# Deflates the matrix A so that we can calculate a new eigenvalue
function deflation(A, x, v)
    y = norm(v)^2
    B = A - x/y*v*v"0027
    return B
end
```

[3]: deflation (generic function with 1 method)

[4]:
```
# Combines max_power_method and deflation to calculate all of the eigenvalues
function power_method(A)
    O = zeros(Float64, size(A))
    A_old = convert(Array{Float64}, A)
    n = size(A)[1]
    ind = 1 # keeps track of how many eigenvalues we have computed
    while(ind <= n)
        x, v = max_power_method(A_old)
        A_new = deflation(A_old, x, v)
        O[ind,ind] = x
        #A_final[:, ind] = v
        A_old = A_new
        ind += 1
    end
    return O
end
```

[4]: power_method (generic function with 1 method)

### QR Decomposition

[5]:
```
# slower method for QR decompositon
# directly writes the Gram-Schmidt algorithm into code
# see "Algorithm 1" in paper
function QR(M)
    Q = Array{Float64, 2}(undef, size(M))
    R = zeros(Float64, size(M)[2], size(M)[2])
    for k = 1:size(M)[2]
        a = M[:, k]
        s = a
        for i = 1:k-1
            s -= dot(Q[:, i], a) * Q[:, i]
        end
        Q[:, k] = s / norm(s)
    end

    for i = 1:size(M)[1]
        for j = i:size(M)[2]
```

```
            R[i, j] = dot(Q[:, i], M[:, j])
        end
    end
    return Q, R
end
```

[5]: QR (generic function with 1 method)

[6]:
```
# faster method of QR decomposition
# see "Algorithm 2" in paper
function QRfast(M)
    Q = Array{Float64, 2}(undef, size(M))
    R = zeros(Float64, size(M)[2], size(M)[2])
    for k = 1:size(M)[2]
        s = dot(M[:, k], M[:, k])
        R[k, k] = sqrt(s)
        Q[:, k] = M[:, k] / R[k, k]
        Q[:, k] = Q[:, k] / norm(Q[:, k])
        for i = k+1:size(M)[2]
            s = dot(M[:, i], Q[:, k])
            R[k, i] = s
            M[:, i] -= R[k, i] * Q[:, k]
        end
    end
    return Q, R
end
```

[6]: QRfast (generic function with 1 method)

[7]:
```
# Used to time the QR and QRfast algorithms
function get_speeds(s)
    k = 100
    t1s = zeros(Float64, k)
    t2s = zeros(Float64, k)
    for i = 1:k
        B = rand(s, s)
        A = transpose(B) * B
        t1 = @timed QR(A)
        t2 = @timed QRfast(A)
        t1s[i] = t1[2]
        t2s[i] = t2[2]
    end
    println(s, "\t", mean(t1s), "\t", mean(t2s))
end
```

[7]: get_speeds (generic function with 1 method)

### QR Method

The first set of QR method implementations only calculate eigenvalues. These were used in testing between different implementations, such as using deflation and shifting (see paper, section 3.2). The last QR method implements eigenvector calculation as well.

[8]:
```
# Simple QR Method for only eigenvalues -- no shifting, no deflation
# see paper, "Algorithm 1"
function eigenvals_naive(A)
    B = copy(A)
    n = size(B)[1]
    eigenvalues = zeros(Float64, n)
    current = B[n, n]
    while(n > 0)
        Q, R = QRfast(B)
        B = R * Q
        if isapprox(B[n, n], current; atol=1e-3, rtol=0)
            eigenvalues[n] = B[n, n]
            n -= 1
        else
            current = B[n, n]
        end
    end
    return eigenvalues
end
```

[8]:
```
eigenvals_naive (generic function with 1 method)
```

[9]:
```
# QR Method for only eigenvalues, implementing deflation -- still no shifting
function eigenvals_deflate(A)
    B = copy(A)
    n = size(B)[1]
    eigenvalues = zeros(Float64, n)
    current = B[n, n]
    while(n > 0)
        Q, R = QRfast(B)
        B = R * Q
        if isapprox(B[n, n], current; atol=1e-3, rtol=0)
            eigenvalues[n] = B[n, n]
            n -= 1
            B = B[1:n, 1:n] #deflate
        else
            current = B[n, n]
        end
    end
    return eigenvalues
end
```

[9]: eigenvals_deflate (generic function with 1 method)

[10]:
```
# formula for Wilkinsion shift
function wilkinson(A, n)
    if n == 1
        return 0 # if problem is occurred, return 0 (no shift)
    end
    x = A[n-1, n-1]
    y = A[n,n]
    z = A[n-1, n]
    delta = (x - z)/2
    if delta >= 0
        sgn = 1
    else
        sgn = -1
    end
    rval = z - (sgn * y * y)/(abs(delta) - sqrt(delta * delta + y * y))
    if rval == -Inf || rval == NaN
        return 0 # if problem is occurred, return 0 (no shift)
    end
    return rval
end
```

[10]: wilkinson (generic function with 1 method)

[11]:
```
# QR Method for only eigenvalues, implementing deflation and shifting
function eigenvals_shift(A)
    B = copy(A)
    n = size(B)[1]
    eigenvalues = zeros(Float64, n)
    current = B[n, n]
    while(n > 1)
        # Rayleigh shift currently implemented, Wilkinsion commented
        mu = B[n, n] #wilkinson(B, n)
        Q, R = qr(B - (mu * I))
        B = (R * Q) + (mu * I)
        # test for convergence:
        if isapprox(B[n, n], current; atol=1e-5, rtol=0)
            eigenvalues[n] = B[n, n]
            n -= 1
            B = B[1:n, 1:n]
        else
            current = B[n, n]
        end
    end
    eigenvalues[1] = B[1, 1]
    #eigenvalues = diag(B)
```

```
    return reverse(sort(eigenvalues))
end
```

[11]: eigenvals_shift (generic function with 1 method)

[12]:
```
# Used to time the three implementations of QR method
function get_speeds(s)
    k = 100
    t1s = zeros(Float64, k)
    t2s = zeros(Float64, k)
    t3s = zeros(Float64, k)
    for i = 1:k
        B = rand(s, s)
        A = transpose(B) * B
        t1 = @timed eigenvals_naive(A)
        t2 = @timed eigenvals_deflate(A)
        t3 = @timed eigenvals_shift(A)
        t1s[i] = t1[2]
        t2s[i] = t2[2]
        t3s[i] = t3[2]
    end
    println(s, "\t", mean(t1s), "\t", mean(t2s), "\t", mean(t3s))
end
```

[12]: get_speeds (generic function with 1 method)

The final QR algorithm, implementing shifting, eigenvector calculation, and inverse iteration to make eigenvector calculation more accurate is shown in ``get_eigens'' below:

[13]:
```
# gets eigenvalues, eigenvectors
# does not implement inverse iteration to hone eigenvector approximation
# helper function of "get_eigens"
function qr_get_eigenvalues(A)
    B = copy(A)
    n = size(B)[1]
    eigenvalues = zeros(Float64, n)
    current = B[n, n]
    Qiter = Matrix{Float64}(I, n, n)
    while(n > 1)
        mu = B[n, n]
        Q, R = qr(B - (mu * I))
        B = (R * Q) + (mu * I)
        if isapprox(B[n, n], current; atol=1e-5, rtol=0)
            eigenvalues[n] = B[n, n]
            n -= 1
            B = B[1:n, 1:n]
        else
```

```
            current = B[n, n]
        end
    end
    eigenvalues[1] = B[1, 1]
    return reverse(sort(eigenvalues)), Qiter
end
```

[13]: qr_get_eigenvalues (generic function with 1 method)

[14]:
```
# helper function for inverse iteration
function inverse_iteration_single(A, val, b)
    try
        c = inv(A - val * I) * b
        return c / norm(c)
    catch e
        return inverse_iteration_single(A, val * 1.5, b)
    end
end
```

[14]: inverse_iteration_single (generic function with 1 method)

[15]:
```
# implements inverse iteration given:
    # matrix A
    # eigenvalue approx. val
    # eigenvector approx. b
# runs n times
function inverse_iteration(A, val, b, n)
    c = copy(b)
    for i = 1:n
        c = inverse_iteration_single(A, val, b)
    end
    return c
end
```

[15]: inverse_iteration (generic function with 1 method)

[16]:
```
# calculates eigenvectors and eigenvalues using QR method and inverse iteration
# see paper, "Algorithm 4"
function get_eigens(A)
    vals, Q = qr_get_eigenvalues(A)
    for i = 1:size(vals)[1]
        Q[:, i] = inverse_iteration(A, vals[i], Q[:, i], 25)
    end
    return vals, Q
end
```

[16]: get_eigens (generic function with 1 method)

[17]:
```
# simple test for eigenvector/eigenvalue approximations
# calculates difference between Ax and 0x
function test_eigens(A, vals, vecs)
    test = zeros(size(vals)[1])
    for i = 1:size(vals)[1]
        println(norm((A * vecs[:, i]) - (vals[i] * vecs[:, i])))
        test[i] = norm((A * vecs[:, i]) - (vals[i] * vecs[:, i]))
    end
    return Statistics.mean(test)
end
```

[17]:
```
test_eigens (generic function with 1 method)
```

### Sparse and Dense Matrices

The first set of methods is used to design test matrices and ensure they functon correctly. See paper, section 4.

[18]:
```
# counts the total number of zeros in a matrix.
function count_zeroes(A)
    total = 0
    for i = 1:size(A)[1]
        for j = 1:size(A)[2]
            if A[i, j] == 0
                total += 1
            end
        end
    end
    return total
end
```

[18]:
```
count_zeroes (generic function with 1 method)
```

[19]:
```
# generates test matrix of dim. "0027size"0027 x "0027size"0027
# matrix must be symmetric, have approx. "0027zero_ratio"0027 percent zeros as␣
 ↪its elements
function generate_matrix(size, zero_ratio)
    zero_count = trunc(Int, (size * size) * zero_ratio * 0.5)
    val_size = trunc(Int, size * (size * 0.5 + 0.5))
    vals = rand(val_size) # total number of distinct values in symmetric matrix
    r = randperm(val_size - size)[1:zero_count] # get elements to be 0
    vals[r] .= 0
    A = zeros(size, size)
    x = 1
    y = 1
    for i = 1:val_size
        A[x, y] = vals[i]
        A[y, x] = vals[i]
```

```
        x += 1
        if x > size
            y += 1
            x = y
        end
    end
    return A
end
```

[19]: generate_matrix (generic function with 1 method)

[20]:
```
# test speed of function between qr method with shifting, deflation, and power␣
 ↪method
# note that power method from IterativeMethod pkg is used (see paper section 4␣
 ↪for explanation)
function get_speeds(size, zero_ratio)
    k = 100
    t1s = zeros(Float64, k)
    t2s = zeros(Float64, k)
    z = zeros(Float64, k)
    for i = 1:k
        A = generate_matrix(size, zero_ratio)
        z[i] = count_zeroes(A)
        t1 = @timed eigenvals_shift(A)
        t2 = @timed powm(A, inverse=false, log=false)
        t1s[i] = t1[2]
        t2s[i] = t2[2]
    end
    println(mean(z) / (size * size), "\t", mean(t1s), "\t", mean(t2s))
end
```

[20]: get_speeds (generic function with 2 methods)

### Singular Value Decomposition

[21]:
```
# gets the singular value decompositon using QR method
# see paper section 5, and paper, algorithm 5
function getSVD(A)
    sigma = zeros(Float64, size(A))
    if size(A)[1] <= size(A)[2]
        SVcount = size(A)[1]
        ATA = transpose(A) * A
        ATA_vals, V = get_eigens(ATA)
        U = zeros(Float64, SVcount, SVcount)
        for i = 1:SVcount
            sigma[i, i] = sqrt(ATA_vals[i])
            U[:, i] = (A * V[:, i]) / sigma[i, i]
```

```
        end
        return U, sigma, V
    else
        SVcount = size(A)[2]
        AAT = A * transpose(A)
        AAT_vals, U = get_eigens(AAT)
        V = zeros(Float64, SVcount, SVcount)
        for i = 1:SVcount
            sigma[i, i] = sqrt(AAT_vals[i])
            V[:, i] = (transpose(A) * U[:, i]) / sigma[i, i]
        end
        return U, sigma, V
    end
end
```

[21]: getSVD (generic function with 1 method)

[22]:
```
# simple testing function for SVD calculation
# calculated difference between matrix and U*Sigma*V^T
function testSVD(A, U, S, V)
    diff = abs.(A - U * S * transpose(V))
    return mean(diff)
end
```

[22]: testSVD (generic function with 1 method)

### Image Compression

[23]:
```
# takes an image and gets matrix of floats describing black and white version␣
 ↪of image
function get_matrix_from_image(img_string)
    img = testimage(img_string) # gets image from Julia TestImages
    gray_img = Gray.(img)
    return convert(Array{Float64}, gray_img)
end
```

[23]: get_matrix_from_image (generic function with 1 method)

[24]:
```
# Calculate a rank r image compression from the SVD
function compress(U, S, V, r)
    Ssmall = copy(S)
    for i = r+1:size(S)[2]
        Ssmall[i, i] = 0
    end
    return U * Ssmall * transpose(V)
end
```

[24]: compress (generic function with 1 method)

[25]:
```
# will load color image of fabio (256x256), perform low rank approx.␣
 ↪compression (rank 50), and present compressed image
# note -- may take some time to run!
function compression_example()
    M = get_matrix_from_image("fabio_color_256")
    println("loaded. getting SVD...")
    U, S, V = getSVD(M)
    println("SVD obtained. getting compression...")
    small = compress(U, S, V, 50) # rank 50 compression
    println("complete!")
    return Gray.(small)
end
```

[25]: compression_example (generic function with 1 method)

[26]:
```
compression_example()
```

```
loaded. getting SVD…
SVD obtained. getting compression…
complete!
```

[26]:

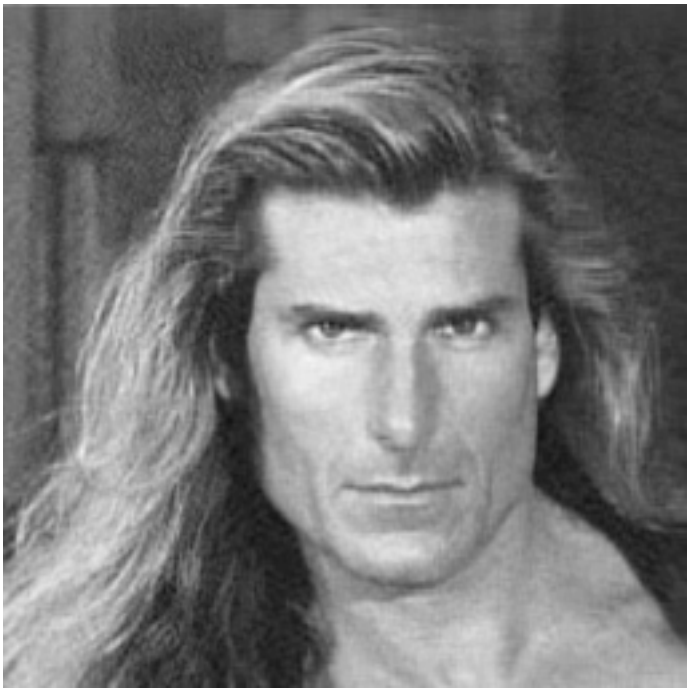


**Image Steganography**

[27]:
```
# Encodes message into image
# M is output of "get_matrix_from_image"
# msg is array of 0"0027s and 1"0027s
function encoder(M, msg)
    code = copy(M)
    msg_index = 1
    for i = 1:size(M)[1]/8 - 1
        for j = 1:size(M)[2]/8 - 1
            submatrix = code[convert(Int64, 8*i):convert(Int64, 8*i+7),␣
↪convert(Int64, 8*j):convert(Int64, 8*j+7)]
            U, s, V = svd(submatrix)
            S = diagm(s)
            # If we want to send a "0"
            if msg[convert(Int64, msg_index)] == 0
                S[8, 8] = 0
                else # If we want to send a "1"
                if abs(S[1, 1]) < 1.0e-10
                    S[5, 5] = .01
                end
                if abs(S[2, 2]) < 1.0e-10
                    S[5, 5] = .01
                end
                if abs(S[3, 3]) < 1.0e-10
                    S[5, 5] = .01
                end
                if abs(S[4, 4]) < 1.0e-10
                    S[5, 5] = .01
                end
                if abs(S[5, 5]) < 1.0e-10
                    S[5, 5] = .01
                end
                if abs(S[6, 6]) < 1.0e-10
                    S[5, 5] = .01
                end
                if abs(S[7, 7]) < 1.0e-10
                    S[5, 5] = .01
                end
                if abs(S[8, 8]) < 1.0e-10
                    S[5, 5] = .01
                end
            end
            code[convert(Int64, 8*i):convert(Int64, 8*i+7), convert(Int64, 8*j):
↪convert(Int64, 8*j+7)] = U * S * transpose(V)
            msg_index += 1
            if msg_index > size(msg)[2]
                return code
            end
```

```
        end
    end
    println("message too long - not fully encoded!")
    return code
end
```

[27]:
```
encoder (generic function with 1 method)
```

[28]:
```
# Decodes message from image, given length of message
# M is output of "get_matrix_from_image"
# returns array of 0"0027s and 1"0027s
function decoder(M, len)
    # M is m x n
    code = copy(M)
    msg_index = 1
    msg = Array{Int64,2}(undef, 1, len)
    for i = 1:size(M)[1]/8 - 1
        for j = 1:size(M)[2]/8 - 1
            submatrix = code[convert(Int64, 8*i):convert(Int64, 8*i+7),␣
 ↪convert(Int64, 8*j):convert(Int64, 8*j+7)]
            U, S, V = svd(submatrix)
            #println(S[8])
            if S[8] < 1e-10
                msg[msg_index] = 0
            else
                msg[msg_index] = 1
            end
            msg_index += 1
            if msg_index > len
                return msg
            end
        end
    end
    println("message len too long - not fully encoded!")
    return msg
end
```

[28]:
```
decoder (generic function with 1 method)
```

[29]:
```
# will load color image of fabio (256x256), make it black and white, hide bit␣
 ↪string, and decrypt bit string
# note -- may take some time to run!
function steganography_example()
    M = get_matrix_from_image("fabio_color_256")
    msg = [0 1 1 1 0 1 0 0 0 1 1 0 1 1 1 1 0 1 1 1 0 0 1 0 0 1 1 0 1 0 0 1 0 1␣
 ↪1 0 1 1 1 0 0 0 0 0 0 0 0 0]
    code = encoder(M, msg)
```

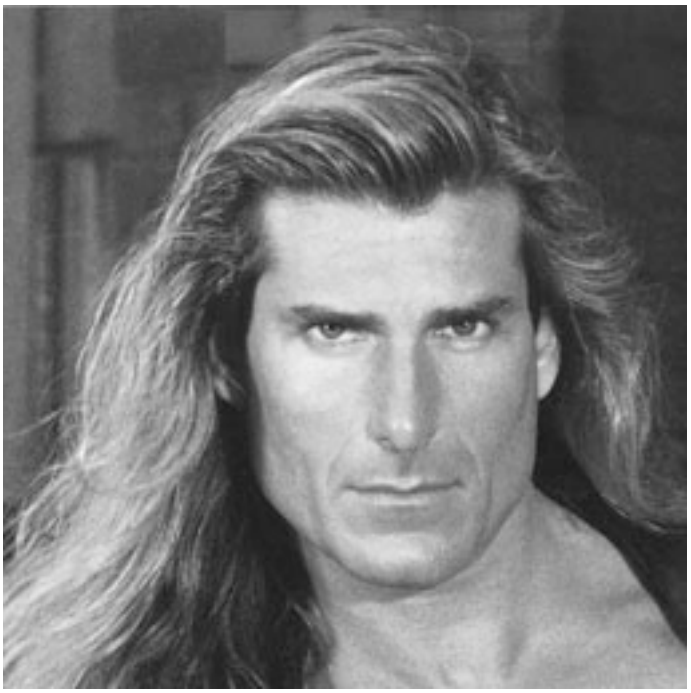
```
    d = decoder(code, 48)
    println("Encoded message:")
    println(msg)
    println("Decoded message:")
    println(d)
    println("Difference:")
    println(abs.(d - msg))
    println("Encoded Fabio:")
    return Gray.(code) # encoded image

end
```

[29]: steganography_example (generic function with 1 method)

[30]:
```
steganography_example()
```

```
Encoded message:
[0 1 1 1 0 1 0 0 0 1 1 0 1 1 1 1 0 1 1 1 0 0 1 0 0 1 1 0 1 0 0 1 0 1 1 0 1 1 1 0
0 0 0 0 0 0 0 0]
Decoded message:
[0 1 1 1 0 1 0 0 0 1 1 0 1 1 1 1 0 1 1 1 0 0 1 0 0 1 1 0 1 0 0 1 0 1 1 0 1 1 1 0
0 0 0 0 0 0 0 0]
Difference:
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0]
Encoded Fabio:
```

[30]:



Thank you!